Proceedings of the 11th International Conference
on Computational and Mathematical Methods
in Science and Engineering, CMMSE 2011
26–30 June 2011.

Integrating dense and sparse data partitioning

Javier Fresno[1], Arturo González-Escribano[1] and Diego R. Llanos[1]

[1] *Dept. Informática, Universidad de Valladolid*

emails: `javfres@gmail.com`, `arturo@infor.uva.es`, `diego@infor.uva.es`

Abstract

Layout methods for dense and sparse data are often seen as two separate problems with its own particular techniques. However, they are based on the same basic concepts. This paper studies how to integrate automatic data-layout and partition techniques for both dense and sparse data structures. In particular, we show how to include support for sparse matrices or graphs in Hitmap, a library for hierarchical-tiling and automatic mapping of arrays. The paper shows that is possible to offer a unique interface to work with both dense and sparse data structures, without losing significant performance. Thus, the programmer can use a single and homogeneous programing style, reducing the development effort and simplifying the use of sparse data structures in parallel computations.

Key words: data partition, layouts, distributed computing, sparse.

1 Introduction

In parallel applications the data distribution and layout is a key issue that can determine the performance and scalability. Regarding the type data structures, dense or sparse, the data distribution problem is treated differently. On the one hand, there are many languages with primitives and tools to deal with data locality and/or distribution, such as HPF [11], OpenMP [4], or UPC [2]. On the other hand, sparse structure support are not usually integrated in the programming languages. However, a wide range of important problems are based in unstructured graph structures instead of dense arrays. The common approach to manage sparse data is using a library. We can find a high number of libraries for partitioning graphs, meshes and other sparse structures, such as Metis [10], Scotch [12], or Jostle [13].

There are some proposals that use an unique representation for different domains. Chapel, a new parallel language, uses a representation of indexes-set called a domain [3]. The Chapel's proposal aims to support domains for dense, sparse, strided, associative, and unstructured data aggregates. However, there is not yet a full nor efficient implementation.

Another previous proposal [9] have evolved into the Trasgo system and the Hitmap library. Hitmap is a basic tool in the back-end of the Trasgo compilation system [8]. Trasgo proposes the use of a global-view approach, with flexible and explicit mechanisms to deal with locality. The code generated by Trasgo uses the Hitmap library to perform a highly efficient data distribution and aggregated communications.

In this paper, we present a new approach to hide the internal details of dense and sparse data structure, using a common interface to deal with both types of data. Combining the dense and sparse manipulation under a common interface has great advantages. It simplifies programming by hiding the partition details in reusable and flexible plug-ins. The programmer focus on the algorithm parallel implementation without thinking in terms of the underlying data structure. Coding a parallel application follows the same basic pattern, using the same API for the same functionalities. The implementaton of Hitmap abstractions is focused on further native compiler optimizations. Our experimental results show that using Hitmap simplifies programing with a negligible impact on performance.

This paper is organized as follows: Section 2 provides a brief overview of the Hitmap library. Section 3 introduces the benchmark that illustrates our proposal and describes its different implementations. Section 4 contains the experimental work. Finally, the paper ends with the conclusions at Section 5.

2 Hitmap library

Hitmap is a highly-efficient library for hierarchical tiling and mapping of arrays [6, 7]. It aims to simplify parallel programming, providing functionalities to create, manipulate, distribute, and communicate tiles and hierarchies of tiles. In this section we will present the basic ideas of the Hitmap library needed for the further discussion.

Hitmap library supports functionalities to: (1) Generate a virtual topology structure; (2) mapping the data to the different processor with chosen load-balancing techniques; (3) automatically determine inactive processors at any stage of the computation; (4) identify the neighbor processors to use in communications; and (5) build communication patterns to be reused across algorithm iterations.

Hitmap is designed with an object-oriented approach, although it is implemented in C language. Fig. 1 shows a class diagram of the library architecture. The classes are implemented as C structures with associated functions. A *Signature*, represented by a HitSig object, is a selection of array indexes in a one-dimensional domain. Hitmap uses a *Shape* object to represents a domain of data. In the previous Hitmap version, this object was composed by HitSig objects, and it represented a contiguous or stride subset of indexes in a multidimensional domain. The new Hitmap version uses inheritance to integrate shape objects that represent sparse data domains. A *Tile* is an array which domain is defined by a shape. Hitmap has functionalities to dynamically declare selections of tiles to construct tile hierarchies. A tile may be defined without allocated memory allowing to declare and partition arrays before assigning memory to them, finally allocating only the parts mapped to a given processing unit.

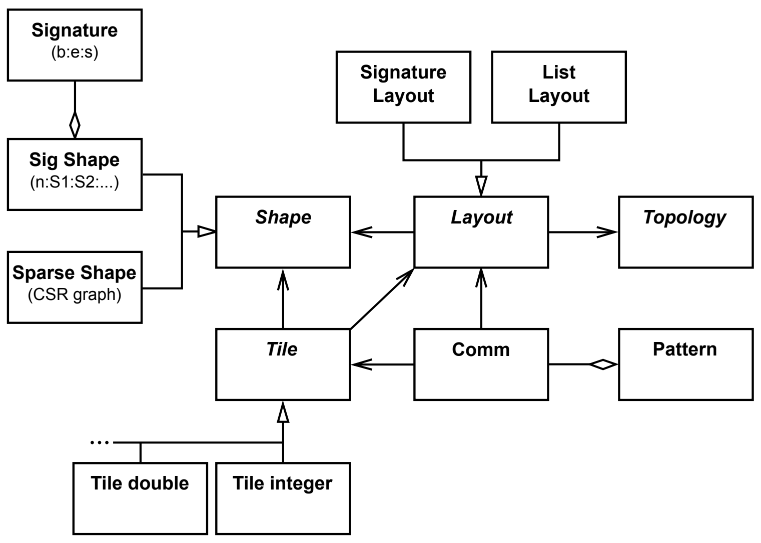


Figure 1: Hitmap new architecture.

Hitmap provides programming tools to apply different data-partition and layout techniques over automatically generated virtual topologies, hidding the details of the physical processors, topology, and mapping. Both the virtual topology generation and the partition techniques are integrated in the library as plug-in modules extending abstract classes. Programmers may include their own new techniques.

The virtual topology techniques are invoked by name with no extra parameters. They use the internal information of the target system. The result is a HitTopology object which can be queried and it is used as a parameter for data partition and layout. The layout plug-in modules allow to compute a partition of a shape domain over a virtual topology. The result of a layout plug-in is a HitLayout object that contains information about the local part of the domain mapped to the current processor, the neighbor relations, and methods to return or compute on the fly all the information needed to exchange data. The plug-ins encapsulate the computations needed to deal with physical topology and data location at all the mapping stages; details which are usually hardwired in the code by the programmer. The combination of these plug-in systems allows the programmer to easily create abstract codes which also simplifies debugging operations with any kind of data structures.

Finally, an information exchange operation can be specified creating a HitCom object. The constructor receives a HitShape to specify the data to be moved and a HitLayout object with the mapping and topology information. The result is a HitCom object with all the information needed to execute the data exchange as many times as required. Moreover, several HitCom objects can be composed in reusable communications patterns represented by HitPattern objects. The library is built on top of the MPI communication library, for portable communication and synchronization on different architectures. Hitmap internally exploits several MPI techniques that increase performance.

2.1 Hitmap Sparse support

The previous version of the Hitmap library was oriented to manipulate and communicate tiles of data with contiguous domains, or non-contiguous but regular index selections. In this new version of the library, we have extended the Hitmap functionalities to support sparse data structures. This allows the programmer to work with graph or sparse matrices using a similar API, and a homogeneous programming methodology. To support the manipulation, mapping, and communication of these new data structures, we need to make several structural changes in the library.

The first change is related to the data domains represented by the HitShape class. In the previous version of the library, a HitShape object was composed by several HitSig objects, to represent a multidimensional selection of indexes. We have transformed the HitShape class into an interface, with two different implementations, one for the old dense domain and another for the new sparse domain (see Fig. 1).

The HitSparseShape class encapsulate a sparse matrix format to represent the sparse data domain. The first sparse matrix format that we have implemented is Compressed Sparse Row (CSR). It is a well-known and widely used format for sparse data. CSR is simple; it does not make any assumptions about the matrix structure and it has minimal storage requirements [1]. There are other formats that can offer a better performance in some particular applications. It is possible to support any of these formats with new implementations of the proposed interface.

To illustrate how to develop new layout plug-ins that make specific partitions and mapping techniques for sparse domains, we have implemented an example HitLayout plug-in. We have integrated one of the graph partition techniques of the Metis library [10]. The plug-in receives a HitShape with the sparse domain, and it calls the Metis library to compute the local part. Metis also uses the CSR format. Thus, the plug-in needs to apply a minimal data-format transformation. With the result returned by Metis, the plug-in creates a new HitShape with the local graph part. The resulting HitLayout object can optionally contain lists with vertices belonging to other processors. This information can be used to find the owner of a given vertex and to exchange data between processors.

To implement simple FDM (Finite Differences Methods) applications, we have included code to automatically compute the list of vertices in other processors which are adjacent to the local vertices. This example plug-in can be further extended to include more neighbors information useful for more complex FEM (Finite Elements Methods) applications.

The tiles constructor receives a HitShape object. There is a method to allocate the memory for the associated data. The tile constructor internally checks the type of shape. For sparse shapes we allocate memory for the vertices. Extending the current implementation to store weight information for the edges is straightforward. We have also added macros and functions to easily access data and iterate across the vertices values.

In some applications, a processor needs the values of neighbor vertices mapped to other processors. The new version of the HitCom object supports a new global communication type. It is designed to exchange the data of adjacent vertices assigned to different processors. The HitCom object uses the internal information calculated in the layout to determine which vertices should be sent to any other processor, and which vertices should the current processor receive from any other processor.

Finally, other functionalities have been added to the library to facilitate the sparse data management. For example, functions for input of sparse data, like reading the Harwell-Boeing format or plain CSR format.

3 Neighbor-vertices synchronization benchmark

In this section we discuss the methodology followed to create a benchmark to test the efficiency of our implementation. We select a simple problem that involves a computation over a sparse data structure.

We extend the idea of neighbor synchronization in FDM applications on dense matrices to civil engineering structural graphs, see e.g. [14]. The application performs several iterations of a graph update operation. It traverses the graph nodes, updating each node value with the result of a function on the neighbor nodes values. To simulate the load of a real scientific application, we write a dummy loop, which issues 10 times a mathematical library operation (sin). We use as benchmarks different graphs from the Pothen group of the *University of Florida Sparse Matrix Collection* [5].

The codes have been run on Geopar, an Intel S7000FC4URE server, equipped with four quad-core Intel Xeon MPE7310 processors at 1.6GHz and 32GB of RAM. The MPI implementation used is MPICH2, compiled with a backend that exploits shared memory for communications if available in the target system.

3.1 C implementation

We have first developed a serial C implementation of the benchmark, to use as the reference to measure speedups and to verify the results of the parallel versions.

Our first parallel version includes the code to compute the data distribution manually. This highly-tuned and efficient implementation is based on Metis library to calculate the partition of the graph, and MPI to communicate the data between processors. The Hitmap library implementation is also based on the same tools. Thus, the comparison between performance results of a Hitmap parallel version and the manual one will show any potential inefficiency introduced in the design or implementation of Hitmap.

The manual parallel version has the following stages: (1) A sparse graph file is read; (2) the data partition is calculated using the Metis library; (3) each processor initializes the values of the local vertices using a random function; (4) a loop performs 10,000 iterations of the main computation, including updating the values of each local

```
// Read the global graph.
HitShape shape_global = hit_shapeHBRead("graph_file.rb");

// Create the topologoy object.
HitTopology topo = hit_topology(plug_topPlain);

// Distribute the graph among the processors.
HitLayout lay = hit_layout(plug_layMetis,topo,&shape_global);

// Get the shapes for the local and local extended graphs.
HitShape local_shape = hit_layShape(lay);
HitShape ext_shape = hit_layExtendedShape(lay);

// Allocate memory for the graph.
HitTile_double graph;
hit_tileDomainShapeAlloc(&graph,sizeof(double),HIT_NONHIERARCHICAL,ext_shape);

// Init the local graph.
init_graph(graph, shape_global);

// Create the communicator and send the initial values of the neighbor vertices.
HitCom com = hit_comSparseUpdate(lay, &graph, HIT_DOUBLE);
hit_comDo(&com);

int i;
// Update loop.
for(i=0; i<ITERATIONS; i++){
   // Update the graph.
   synchronization_iteration(local_shape,ext_shape,&graph);
   // Communication.
   hit_comDo(&com);
}
```

Figure 2: Kernel code of the Hitmap version.

vertex, and the communication of the values for neighbor vertices to other processors; (6) the final result is checked with the help of a hash function.

3.2 Hitmap implementation

Using the manual C implementation as a starting point, we have developed a Hitmap version of the program. The Hitmap implementation uses the main computation and other sequential parts of the previous one, adapting them to work with Hitmap functions for accessing data structures. A new layout plug-in module has been developed to apply the Metis data partition to the Hitmap internal sparse-shape structures. Data-layout and communications have been generated using Hitmap functionalities. In this section we discuss the Hitmap techniques needed to automatically compute the data-layout, allocate the proper part of the graph and communicate the neighbor vertices values.

```
int vertex, edge;

// Iterate through all the vertices.
hit_sparseShapeVertexIterator(vertex,local_shape){

   // Set new value to 0.
   double value = 0;

   hit_sparseShapeEdgeIterator(edge,ext_shape,vertex){

      // Get the neighbor.
      int neighbor = hit_sparseShapeEdgeTarget(ext_shape,edge);
      // Add its contribution.
      value +=  hit_tileElemAt(1,graph,neighbor);
   }

   // Dummy workload = 10.
   int i;
   for(i=0; i<WORKLOAD; i++){
      value = sin(value+1);
   }

   int nedge = hit_sparseShapeNumberEdgesFromVertix(ext_shape,vertex);

   // Update the value of the vertex.
   hit_tileElemAt(1,graph_aux,vertex) =  (value / nedge);
}
```

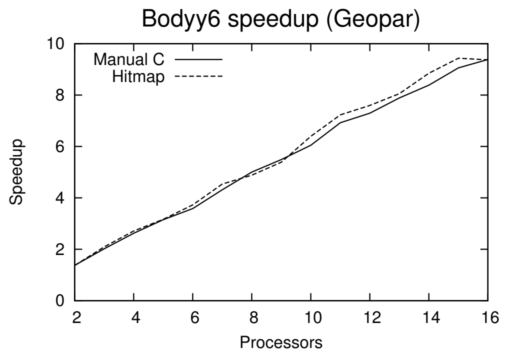
Figure 3: Function that serially updates the local part of the graph.

In Fig. 2 we show the main function of the Hitmap code. The first line initializes the Hitmap environment. Line 5 uses a function to read a graph stored in the file system, and returns a shape object. Then, a virtual topology of processors that uses the internal information available about the real topology is created transparently to the programmer with a single call.

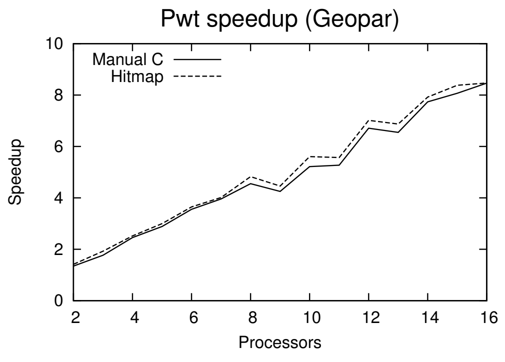
In line 8, the data-layout is generated with a single Hitmap call. The layout parameters are: (a) the layout plug-in name, (b) the virtual topology of processors generated previously, and (c) the shape with the domain to distribute. The result is a HitLayout object, containing the shape assigned to the local processor and information about the neighbors.

In line 11, we obtain the shape of the local part of the graph with only the local vertices. On the following line, we use the layout to obtain an extended shape with local vertices plus the neighbor vertices from other processors. This is the equivalent to the shape of a tile with a shadow region in a FDM solver for dense matrices. This shape is used to declare and allocate the local tile with double elements (line 16).

In line 22, a HitCom object is created to contain the information needed to issue the communications that will update the neighbor vertices values. Data marshaling and



(a) Speedup for the bodyy6 graph.



(b) Speedup for the pwt graph.

Figure 4: Speedup in Geopar.

unmarshaling is automatized by the communications objects when the communication is invocated (see lines 23 and 31).

Lines 25 to 32 contain the main iteration loop to update local nodes values and reissue communications with a single Hitmap call thar reuses the HitCom object previously defined.

Fig. 3 show the code of the function that updates the local values of the graph. It uses two iterators defined in the library. The first one traverses the local vertices (line 4), and the second one iterates over the edges to get their neighbor vertices (line 9). Each neighbor-vertex value contributes to the new value of the local vertex that is set in line 20.

4 Experimental results

In this section we compare the performance obtained with the benchmark described in the previous section. We have tested the benchmark with different graphs from the Pothen group of the *University of Florida Sparse Matrix Collection* [5]. In this section, we discuss the results from two representative cases in the group collection: The bodyy6 graph with 19366 vertices and pwt graph with 36519 vertices. Fig. 4 shows the speedup for both manual C and Hitmap benchmark implementations. There is no significant difference between the two implementations in terms of performance. Therefore, the abstractions introduced by Hitmap (such as the common interface for dense and sparse data structures, or the adaptation of the partition technique in the plug-in module system), do not lead to performance reduction comparing with the manual version.

Fig. 5 shows a comparison of the Hitmap version with the C version in terms of lines of code. We distinguish lines devoted to sequential computation, declarations, parallelism (data layouts and communications), and other non-essential lines (input-output, etc). Taking into account only essential lines, our results show that the use of Hitmap library leads to a 72% reduction on the total number of code lines with respect

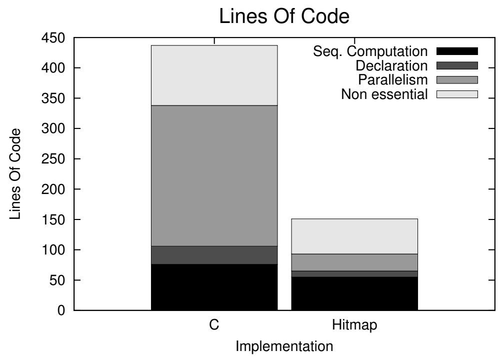


Figure 5: Comparison of the lines of code.

to C version. Regarding lines devoted specifically to parallelism, the percentage of reduction is 78.7%. We have also use the cyclomatic complexity metric to compare the codes. The total cyclomatic complexity of the manual version is 74 whereas the Hitmap versions has a total value of 17. The reason for the reduction of complexity in the Hitmap version is that Hitmap greatly simplifies the programmer effort for data distribution and communication, compared with the equivalent code to manually calculate the information needed in the MPI routines.

5 Conclusions

This paper studies how to integrate the support for dense and sparse data structures in an automatic data partitioning parallel library. We have add sparse structure support in Hitmap, a highly-efficient modular library for hierarchical tiling and mapping of arrays. We have illustrated how to use the library to implement a simple graph algorithm. We have also measured the efficiency of the library in terms of performance comparing with a manual implementation. The results show that the abstraction introduced by the library does not reduce performance. We also measure the code complexity in terms of lines of code and cyclomatic complexity. Our results show that it is possible to use a common interface for both dense and sparse data structures with a homogeneous coding style, and reducing the associated development cost comparing with manually coding the data structure management, its partition, and the communication of locally mapped subdomains when needed. As it is shown by the experimental results, this can be done without sacrificing performance.

Our ongoing work includes the integration of new partition techniques in the Hitmap framework. For example, there are other libraries that could be used instead of the Metis library, with different partitioning properties. We are also working on alternative implementations for the communication classes, that are currently built on top of the MPI communication library, to better exploit different low-level parallel tools and models.

Acknowledgements

This research is partly supported by the Ministerio de Industria, Spain (CENIT MARTA, CENIT OASIS, CENIT OCEANLIDER), Ministerio de Ciencia y Tecnología (CAPAP-H3 network, TIN2010-12011-E), and the HPC-EUROPA2 project (project number: 228398) with the support of the European Commission - Capacities Area - Research Infrastructures Initiative.

References

[1] Richard Barrett, Michael Berry, Tony. F. Chan, James Demmel, June M. Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*, volume 64. SIAM, July 1995.

[2] William W. Carlson, Jesse M. Draper, David E. Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. Introduction to UPC and Language Specification, 1999.

[3] Bradford L Chamberlain, Steven J Deitz, David Iten, and Sung-Eun Choi. User-defined distributions and layouts in chapel: philosophy and framework. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, HotPar'10, page 12, Berkeley, CA, USA, 2010. USENIX Association.

[4] Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel programming in OpenMP*. Morgan Kaufmann, 1 edition, 2001.

[5] Timothy A. Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *To appear in ACM Transactions on Mathematical Software.*

[6] Carlos de Blas Cartón, Arturo González-Escribano, and Diego R. Llanos. Effortless and Efficient Distributed Data-Partitioning in Linear Algebra. In *2010 IEEE 12th International Conference on High Performance Computing and Communications (HPCC)*, pages 89–97. IEEE, September 2010.

[7] Javier Fresno, Arturo González-Escribano, and Diego R. Llanos. Automatic Data Partitioning Applied to Multigrid PDE Solvers. In *2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 239–246. IEEE, February 2011.

[8] Arturo González-Escribano and Diego R. Llanos. Trasgo: a nested-parallel programming system. *The Journal of Supercomputing*, December 2009.

[9] Arturo González-Escribano, Arjan J.C. van Gemund, Valentín Cardeñoso Payo, Raúl Portales-Fernández, and Jose A. Caminero-Granja. A preliminary nested-parallel framework to efficiently implement scientific applications. *High Per-*

formance Computing for Computational Science-VECPAR 2004, pages 541–555, 2005.

[10] George Karypis and Vipin Kumar. MeTiS–A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices–Version 4.0, 1998.

[11] Ken Kennedy, Charles Koelbel, and Hans Zima. The rise and fall of High Performance Fortran. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages - HOPL III*, pages 7.1–7.22, New York, New York, USA, 2007. ACM Press.

[12] François Pellegrini. PT-Scotch and libScotch 5.1 User's Guide, 2010.

[13] Chris Walshaw. The serial JOSTLE library user guide : Version 3.0, 2002.

[14] Barry Wilkinson and Michael Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers.* Prentice Hall, 2004.